Linked list cheatsheet for coding interviews

👋

Introduction

Like arrays, a linked list is used to represent sequential data. It is a linear collection of data elements whose order is not given by their physical placement in memory, as opposed to arrays, where data is stored in sequential blocks of memory. Instead, each element contains an address of the next element. It is a data structure consisting of a collection of nodes which together represent a sequence.
In its most basic form, each node contains: data, and a reference (in other words, a link) to the next node in the sequence.
Advantages
Insertion and deletion of a node in the list (given its location) is O(1) whereas in arrays the following elements will have to be shifted.
Disadvantages
Access time is linear because directly accessing elements by its position in the list is not possible (in arrays you can do `arr[4]` for example). You have to traverse from the start.

Learning resources

- Readings
    - What's a Linked List, Anyway? [Part 1], basecs
    - What's a Linked List, Anyway? [Part 2], basecs
- Videos
    - Singly-linked lists, University of California San Diego
    - Doubly linked lists, University of California San Diego

Types of linked lists

Singly linked list

A linked list where each node points to the next node and the last node points to `null`.

Doubly linked list

A linked list where each node has two pointers, `next` which points to the next node and `prev` which points to the previous node. The `prev` pointer of the first node and the `next`pointer of the last node point to `null`.

Circular linked list

A singly linked list where the last node points back to the first node. There is a circular doubly linked list variant where the `prev` pointer of the first node points to the last node and the `next` pointer of the last node points to the first node.

Implementations

Out of the common languages, only Java provides a linked list implementation. Thankfully it's easy to write your own linked list regardless of language.

Language	API
C++	N/A
Java	`java.util.LinkedList`
Python	N/A
JavaScript	N/A

Time complexity

Operation	Big-O	Note
Access	O(n)	
Search	O(n)	
Insert	O(1)	Assumes you have traversed to the insertion position
Remove	O(1)	Assumes you have traversed to the node to be removed

Common routines

Be familiar with the following routines because many linked list questions make use of one or more of these routines in the solution:

- Counting the number of nodes in the linked list
- Reversing a linked list in-place
- Finding the middle node of the linked list using two pointers (fast/slow)
- Merging two linked lists together

Corner cases

- Empty linked list (head is `null`)
- Single node
- Two nodes
- Linked list has cycles. Tip: Clarify beforehand with the interviewer whether there can be a cycle in the list. Usually the answer is no and you don't have to handle it in the code

Techniques

Sentinel/dummy nodes

Adding a sentinel/dummy node at the head and/or tail might help to handle many edge cases where operations have to be performed at the head or the tail. The presence of dummy nodes essentially ensures that operations will never be done on the head or the tail, thereby removing a lot of headache in

writing conditional checks to dealing with null pointers. Be sure to remember to remove them at the end of the operation.

Two pointers

Two pointer approaches are also common for linked lists. This approach is used for many classic linked list problems.

- Getting the k^{th} from last node - Have two pointers, where one is k nodes ahead of the other. When the node ahead reaches the end, the other node is k nodes behind
- Detecting cycles - Have two pointers, where one pointer increments twice as much as the other, if the two pointers meet, means that there is a cycle
- Getting the middle node - Have two pointers, where one pointer increments twice as much as the other. When the faster node reaches the end of the list, the slower node will be at the middle

Using space

Many linked list problems can be easily solved by creating a new linked list and adding nodes to the new linked list with the final result. However, this takes up extra space and makes the question much less challenging. The interviewer will usually request that you modify the linked list in-place and solve the problem without additional storage. You can borrow ideas from the Reverse a Linked List problem.

Elegant modification operations

As mentioned earlier, a linked list's non-sequential nature of memory allows for efficient modification of its contents. Unlike arrays where you can only modify the value at a position, for linked lists you can also modify the `next` pointer in addition to the `value`.
Here are some common operations and how they can be achieved easily:

- Truncate a list - Set the `next` pointer to `null` at the last element
- Swapping values of nodes - Just like arrays, just swap the value of the two nodes, there's no need to swap the `next` pointer

- Combining two lists - attach the head of the second list to the tail of the first list

Essential questions

These are essential questions to practice if you're studying for this topic.

- Reverse a Linked List
- Detect Cycle in a Linked List

Recommended practice questions

These are recommended questions to practice after you have studied for the topic and have practiced the essential questions.

- Merge Two Sorted Lists
- Merge K Sorted Lists
- Remove Nth Node From End Of List
- Reorder List